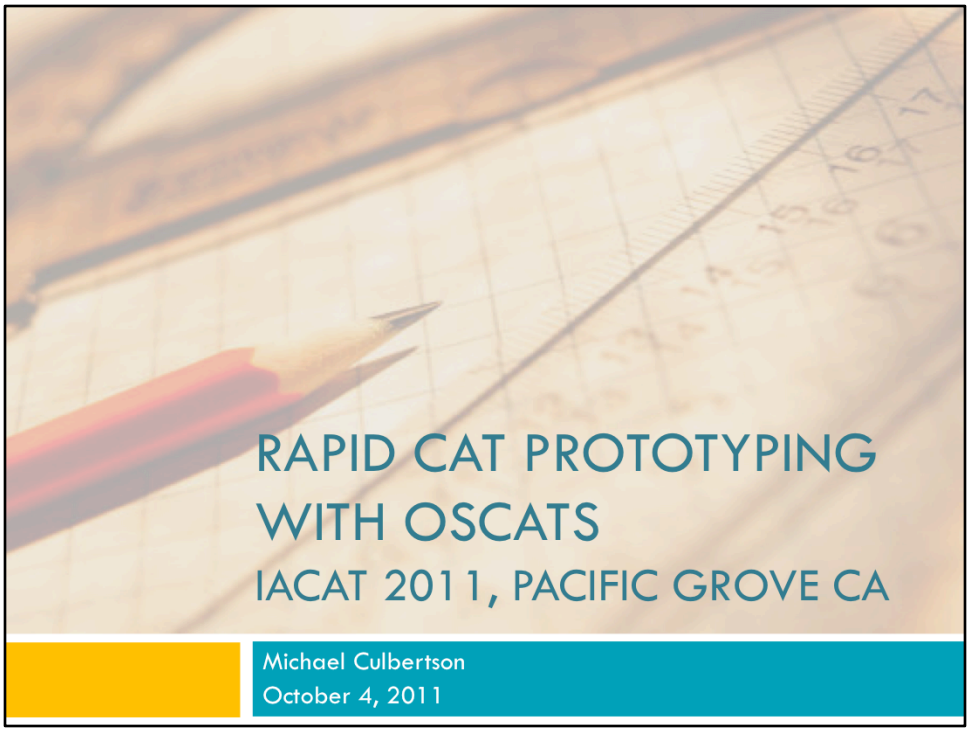
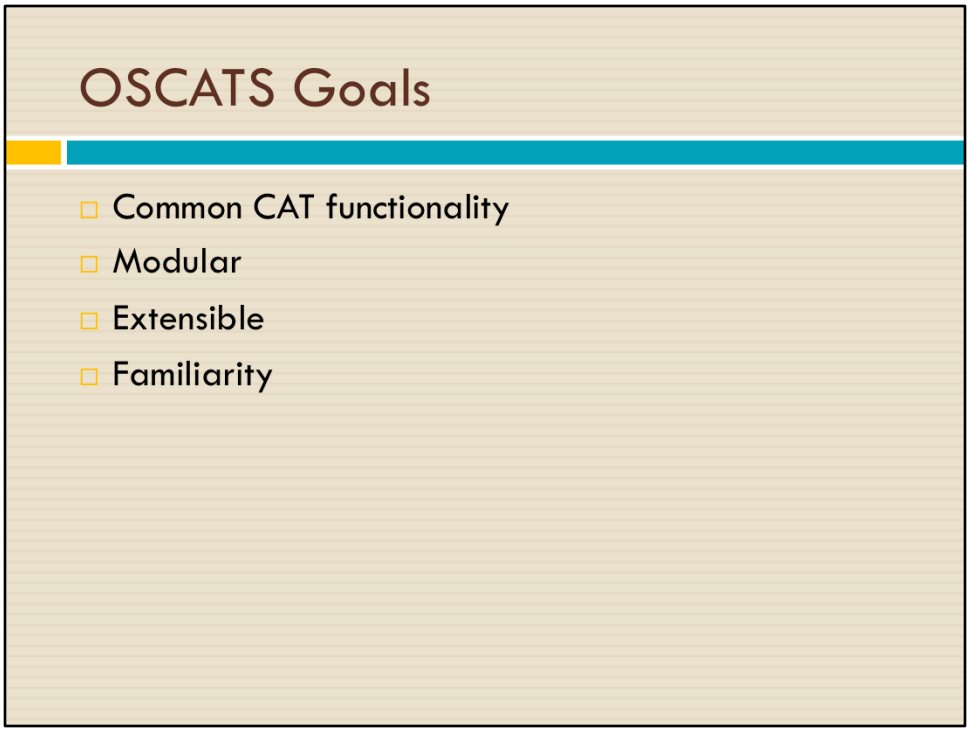


You probably wouldn't be here if you didn't believe that Computerized Adaptive Testing should take a prominent role in the future of measurement. With the benefits of improved efficiency, less-discouraging (or boring) tests for examinees, more-complete item bank usage, and so on, one might wonder why CAT hasn't caught on faster. The reasons behind the seemingly slow adoption of adaptive testing are many, but the one hindrance I would like to focus on today is the relative lack of readily available CAT software. Most current CAT systems are either home-grown or proprietary, meaning that newcomers to the field find themselves in the position of either needing to develop their own software from scratch (assuming they're strong computer programmers) or pay licensing fees, which may be prohibitive for smaller scale innovation. A free, off-the-shelf software library for CAT could allow more developers to begin working with adaptive testing without the very redundant burden of writing, testing, and debugging computer code for common CAT functions. Such a library would streamline entry into CAT research and development, and would free researchers to focus on their substantive questions of interest, instead of wasting time trying to diagnose why their Fisher Information is exploding instead of maximizing.

Today, I would like to present to you a relatively new, free and open source software library that hopes to reduce the burden of getting started in CAT development. It's called, simply enough, the Open Source Computerized Adaptive Testing System, or OSCATS for short. I'll describe how the system is put together; then we'll take a look at how to use OSCATS to construct a CAT simulation to answer a practical research question.



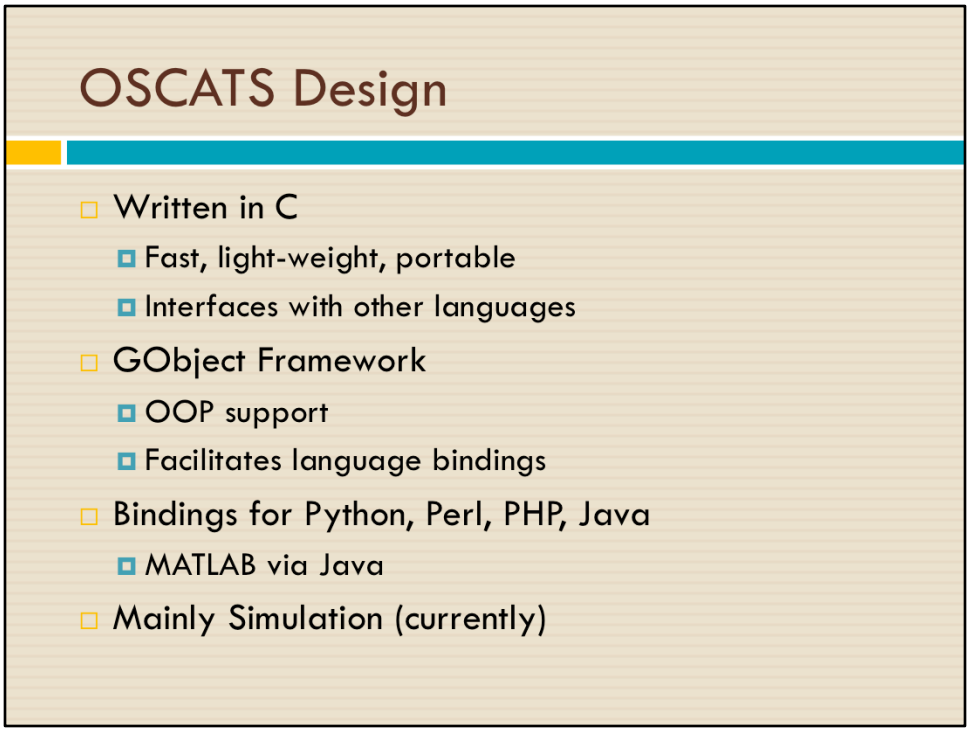
Before delving into the details, I'd like to mention a few of the desired characteristics of a CAT programming library that drive OSCATS development, all of which are intended to facilitate rapid CAT development for research or operational prototyping.

First, OSCATS aims to provide common CAT routines—popular item response models, item selection algorithms, and statistical analysis tools often used to study the properties of an adaptive test. Some of these have already been implemented for the most common choices, and more are under development.

Second, OSCATS is designed to be modular. It should be easy to plug in and pull out or mix-and-match different features of the CAT, such as models and algorithms. To this end, OSCATS is written under the object-oriented programming paradigm. Features such as models and algorithms are represented as objects. These objects define how the CAT features behave in standardized ways so that one IRT model, say, can be easily replaced with another in one location without having to modify all the code that makes use of the model.

Third, OSCATS is designed to be extensible. Although OSCATS already provides some of the most common CAT components and aims to eventually have a very broad selection of models and algorithms, CAT developers should have the freedom to come up with their own twist on a model or algorithm and have it work easily and relatively seamlessly with the rest of OSCATS.

Finally, OSCATS aims to be familiar in the sense of allowing developers to work in a programming language that they already know. Programmers don't all have the same experience, and it would be somewhat limiting to force CAT developers to all use the same language that OSCATS was written in. So, OSCATS provides bindings in other languages; that is, OSCATS comes with a minimal amount glue that allows users to manipulate OSCATS from several different programming languages, allowing programmers to work in their most familiar language.

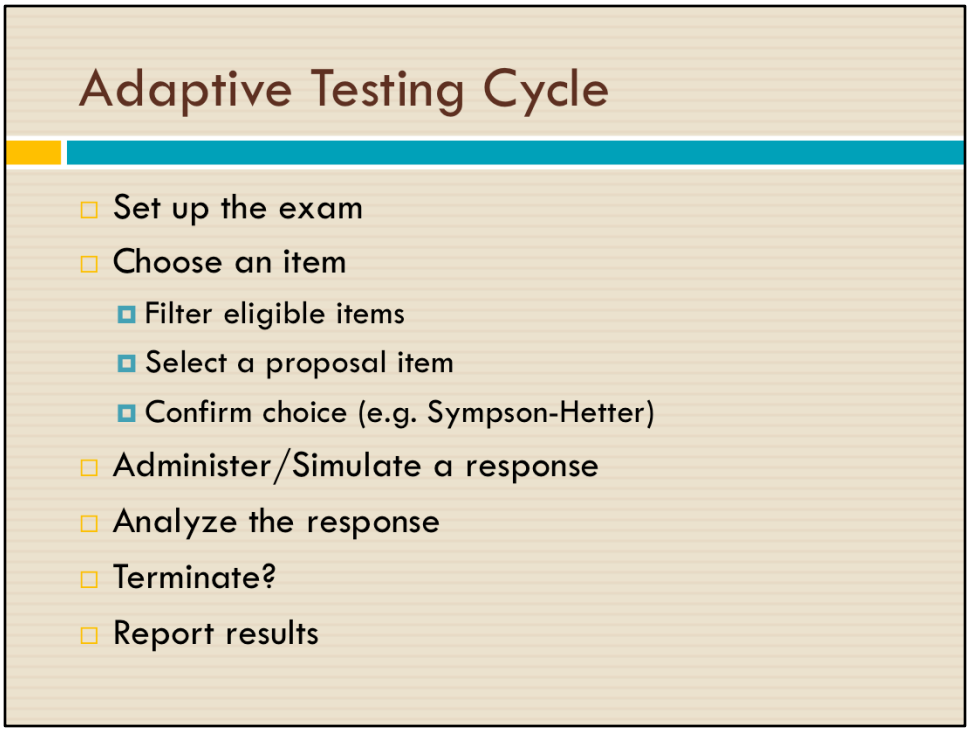


So, which language does OSCATS use? C. Now, given some of the goals I just mentioned, C may not seem like the best choice. C places much more of the burden of memory management on the programmer, which can make debugging more challenging, and its static typing system and compilation requirements don't usually make people think "rapid" and "flexible." But, C was a reasonable choice for several reasons: it's fast, light-weight, portable, and nearly ubiquitous. But more importantly, it is (relatively) straightforward to interface most programming languages out there with libraries written in C. It's not at all straightforward to write a program in Python and manipulate it from PHP, but you can fairly well through C.

Now, you may be thinking that C isn't an object-oriented programming language, like C++ or Java. But, object-oriented programing is really a philosophy or a paradigm, not an intrinsic feature of a programming language. It just so happens that some languages (like C++ and Java) have syntactical features that make using the object-oriented paradigm more convenient. Since the C language doesn't provide much built-in help for object orientation, OSCATS makes use of the GObject framework. Perhaps you've heard of GTK+ (the GUI toolkit) or GIMP (image manipulation program)—both of these are written in object-oriented C using GObject. Now, to be honest, programming with GObject can be a little cumbersome. However, GObject is invaluable for OSCATS's goal of familiarity because it was explicitly designed with language bindings in mind. That is, GObject has certain built-in features that make it easy to write the glue that allows you to use a C library from other languages. Moreover, GObject bindings already exist for over a dozen languages, making it even easier to generate bindings for the OSCATS library.

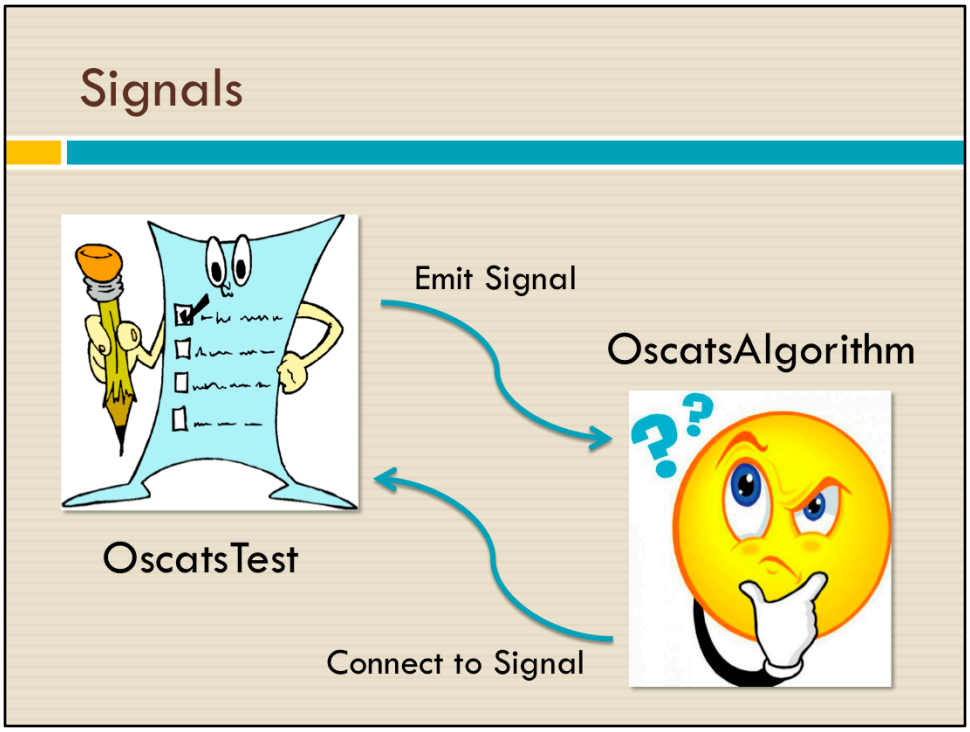
In fact, OSCATS bindings already exist for Python, Perl, PHP, and Java, and it's possible to access OSCATS in MATLAB through the Java bindings. So, even though C and GObject themselves can be somewhat cumbersome, OSCATS users don't actually have to deal with them directly, but can use a more convenient, higher-level language for ease of development.

Finally, I would mention that OSCATS is currently geared toward CAT simulation. That is, it doesn't include a user interface for administering items to actual examinees. OSCATS certainly could be used as the psychometric engine behind such a system in the future, but this just hasn't been the focus of the project so far. (I believe we'll be hearing about another project later in the session that provides a test administration interface for live examinees.) Instead, the flexibility of OSCATS is designed to facilitate research and development of CAT methods and the rapid prototyping of potential operational adaptive tests—for example, studying the properties of a particular item bank or determining the effect of different content balancing techniques and the like.



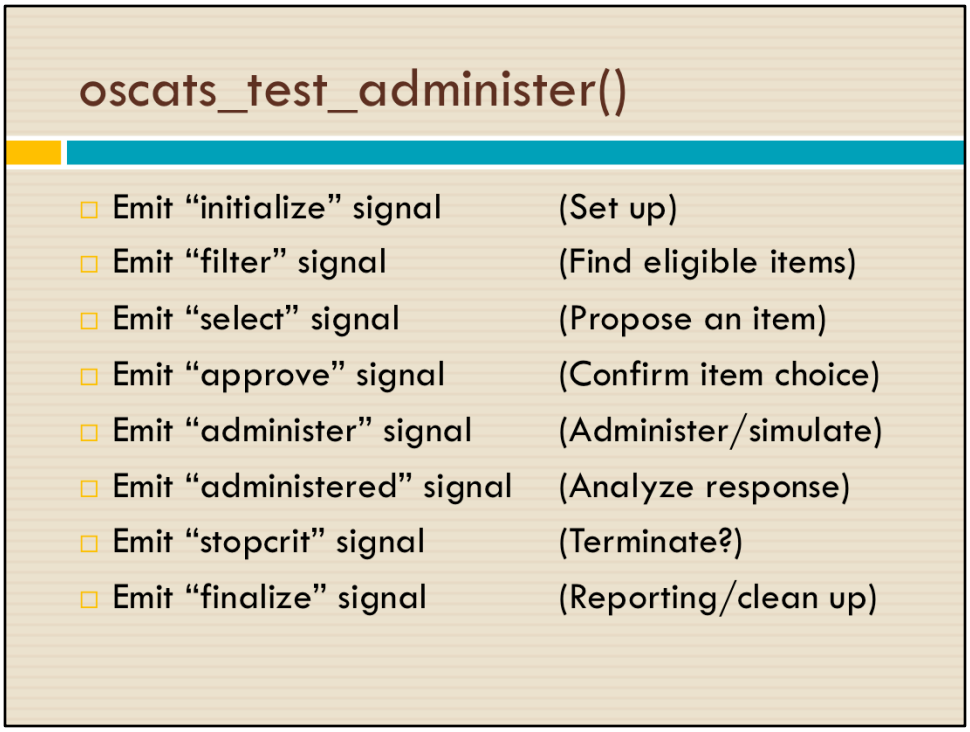
To begin to get into the details of how OSCATS works, let's first think about how adaptive tests run in general. I know you're all quite familiar with this, so I won't spend much time here. A generic CAT cycle starts with some initialization. Then, the CAT chooses an item—this might involve filtering out ineligible items (say in multistage testing or for certain content balancing algorithms), selecting a proposal item (with Maximum Fisher Information or what have you), and potentially approving the choice (e.g. in a SH or content balancing algorithm). Then the item is administered (or simulated), and the response is analyzed (inter alia obtaining a new ability estimate). Then you check a termination criterion and loop, or if the test is complete, you report the results and any associated statistics.

All adaptive tests follow this same basic outline; however, the specific details of each step will depend on the particular algorithms chosen---some might use MFI, others KLI, some have content balancing, others include non-statistical constraints. Usually, a CAT programmer would have to customize the code that implements the adaptive testing cycle by hand to include or exclude algorithms according to the particular design of the given test, which could potentially get a bit messy if a researcher is trying to add and subtract a number of different algorithms to determine their effect on CAT performance. In OSCATS, on the other hand, there is a single piece of code that implements the adaptive testing cycle for *all* OSCATS tests.



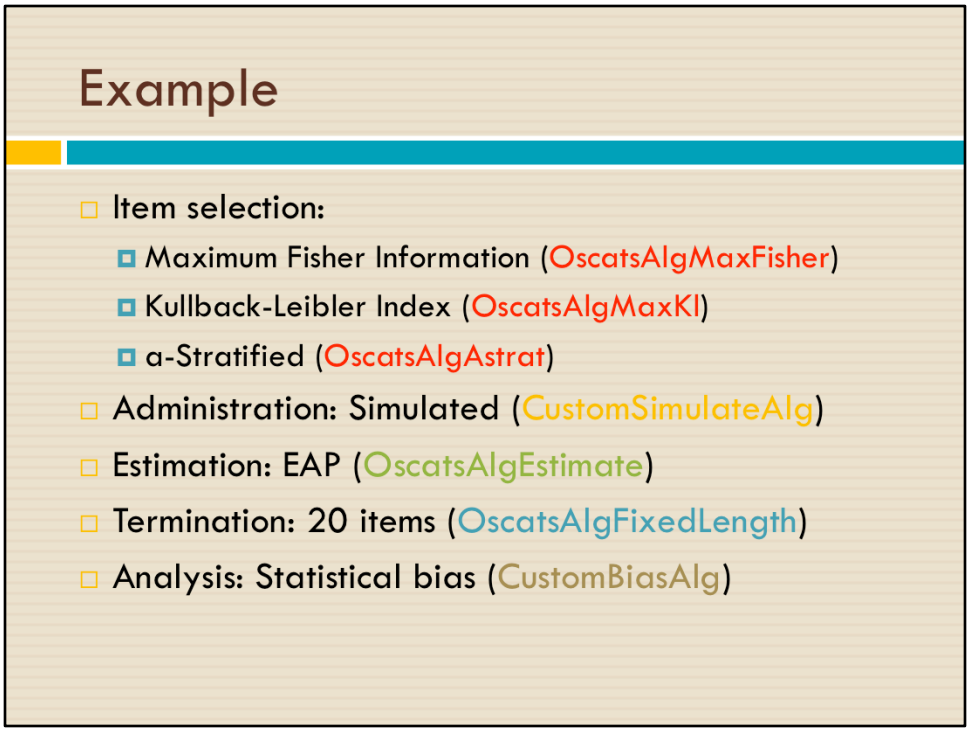
This is achieved using something called signals. A signal is in essence a way for one object to broadcast to other objects that something has happened. Signals are often used in event-based programming, particularly for systems such as graphical user interfaces that have to execute code in an unknown, arbitrary order (namely, in the order that the user click buttons). One object will advertise that it provides a certain signal, and other object have the opportunity to "connect" to the signal for that object. Then, the first objects will send a message (the signal) to all objects that have "signed up" to receive the signal whenever the given event occurs.

Since OSCATS is object oriented, there is a Test object that includes a reference to the item bank and coordinates the administration of the test. There are also "algorithm" objects. It may be a bit strange to think of an algorithm as an object, since algorithms aren't tangible things. But these algorithm objects simply encapsulate the code (and any operating parameters) necessary to perform one piece of the CAT (such as the item selection via MFI, or simulating a response, or estimating ability via the MLE). Setting up an OSCATS test involves creating the Test object and an Algorithm object for each component to be included in the test. Then the Algorithms connect to various signals offered by the Test that correspond to each step of the adaptive testing cycle.



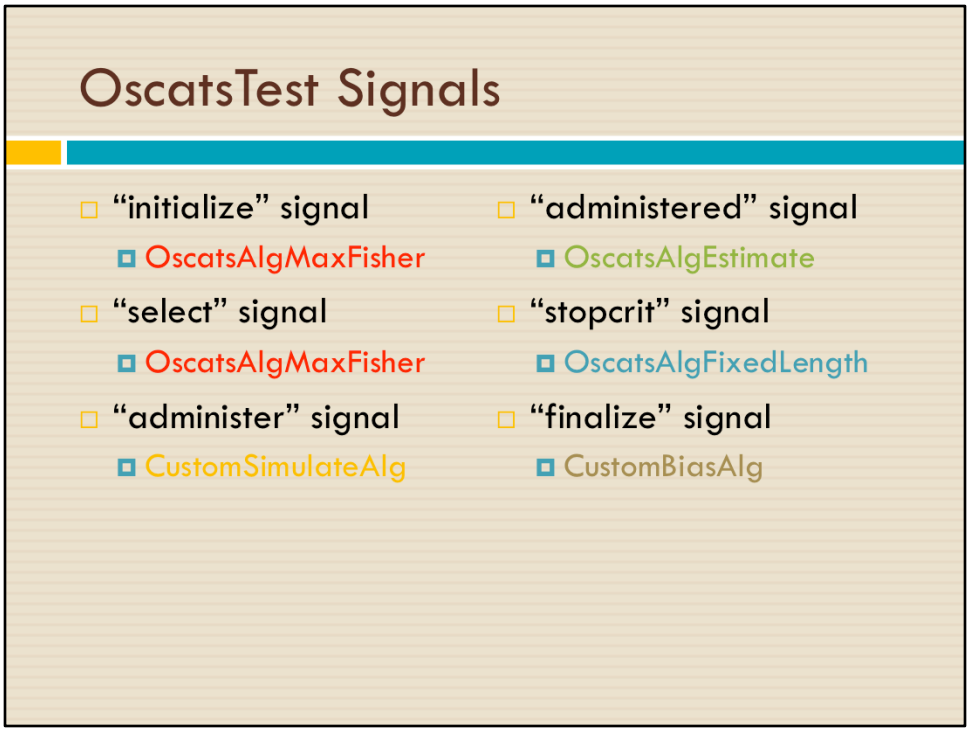
This is how OSCATS achieves modularity. The Test object doesn't actually need to know what algorithms are available to it, or even which algorithms have connected to its signals—all of that is handled automatically by the GObject type system.

The Test simply iterates through each of the stages of the adaptive testing cycle, emitting the appropriate signals in turn. The code that controls OSCATS test administration really isn't much more complicated than what's listed here (plus a little bookkeeping). Changing the particular set of algorithms from one test to the next is simply a matter of connecting a different set of Algorithm objects to the Test. Moreover, writing new algorithms is simplified, since the Algorithm object only has to focus on its one task and "plug in" to the Test object at the appropriate points.



Let's see how this would work for an example CAT. It's a known phenomenon (e.g. Rulison and Loken APM-2009-33-83) that under MFI item selection, examinees who happen to answer the first few of items incorrectly have difficulty recovering, even if they have a high ability level. Suppose you're interested in examining bias in estimated ability under various item selection conditions, but only for those items who answer the first few items either all correct or all incorrect. You'd choose something like this set of OSCATS algorithms.

OSCATS comes with an algorithm to simulate examinee responses, but here we want to do something special (for the first two responses, we want to specify that the examinee gets them both correct or incorrect), so we'll have to write our own administration algorithm, CustomSimulateAlg. But, this isn't too difficult since OSCATS was designed for extensibility. Moreover, even though OSCATS is written in C, we'll work through this example in Python (including writing the new item administration/simulation algorithm).



As we'll see in a moment, OSCATS test construction involves creating an instance of each of these objects and "registering" them for a Test object. On registration, the individual Algorithms take care of connecting themselves to the appropriate signals of the Test object.

OscatsAlgMaxFisher connects to the initialize signal (to initialize some working memory) and the select signal, etc.

As the Test proceeds through the adaptive testing cycle, it emits each signal in turn, which calls the appropriate code for whichever algorithm is connected to the signal. Each Algorithm object only has to be concerned with performing its intended function when its signal is emitted. In this example, we only have one algorithm connected to each signal; but, in principle, any number of algorithms could be connected to a signal (though for some signals, such as "administer," it really only makes sense to have one).

Creating the CAT (Python)

```
test = oscats.Test(id="FI", itembank=mybank,
                   length_hint=20)
oscats.AlgMaxFisher(num=5).register(test)
oscats.CustomSimulateAlg(high=True).register(test)
oscats.AlgEstimate(posterior=True).register(test)
oscats.AlgFixedLength(len=20).register(test)
fi_result = CustomBiasAlg()
fi_result.register(test)
```

First step is to create the Test object. Provide a helpful identifier, indicate the item bank (this code snippet assumes a variable mybank has already been created), and give the test a hint of about how long the test will be (just preallocates some arrays).

Then, create each of the Algorithm objects in turn and have them register with our Test object. Here, have MaxFisher select randomly between the top 5 items. Our custom simulation algorithm is told to force the first two responses correct. Use EAP. Specify the length as 20. In the last case, we store the CustomBiasAlg in the variable fi_result in order to access the results after the simulation.

Creating the CAT (Python)

```
test = oscats.Test(id="KL", itembank=mybank,
                   length_hint=20)
oscats.AlgMaxKl(num=5).register(test)
oscats.CustomSimulateAlg(high=True).register(test)
oscats.AlgEstimate(posterior=True).register(test)
oscats.AlgFixedLength(len=20).register(test)
kl_result = CustomBiasAlg()
kl_result.register(test)
```

Now, if we want to change the configuration from MFI to KLI, due to the modularity of OSCATS, it's simply a matter of replacing one line.

Similarly, notice that we didn't have to tell the algorithms (such as for simulation and estimation) what kind of IRT models we're using. There are objects for the model of each item (created by the programmer elsewhere and stored in the mybank variable, here), and every kind of Model object follows a standard interface, so algorithms can be more or less ignorant of the details of any particular model. They simply ask a model "given this latent ability, what's the probability of answering correctly" (or the like) and leave the details of the implementation to the model object. If we wanted to switch from a 2PL to a 3PL, we don't have to touch any of the algorithm code—we simply modify the section that sets up the items (simulates random parameters or reads parameters from a file—not shown here).

CustomSimulateAlg Implementation

```
class CustomSimulateAlg (oscats.PyAlgorithm) :
  num = gobject.property(type=int, default=2)
  high = gobject.property(type=bool, default=True)

  def __reg__ (self, test) :
    test.connect("administer",
                 CustomSimulateAlg.administer, self)
```

We still have to implement our custom simulation algorithm. We'll look at the full python implementaiton of CustomSimulateAlg over the next 3 slides.

CustomSimulateAlg.administer

```
def administer(self, examinee, item) :
  model = item.get_model(0)
  if (examinee.num_items() < self.num) :
    if (self.high) : resp = 1
    else           : resp = 0

# continued…
```

This is the heart of the algorithm—the piece of code that will be called each time Test emits the administer signal.

CustomSimulateAlg.administer

```
else :   # Simulate as usual
  p = model.P(1, examinee.get_sim_theta(),
              examinee.get_property("covariates"))
  if (oscats.oscats_rnd_uniform() < p) : resp = 1
  else                                 : resp = 0
examinee.add_item(item, resp)
return resp
```

Items and Examinees

- 1,000 Items, 3PL
  - a ~ Unif(0.5, 2.0)
  - b ~ Unif(-3, 3)
  - c ~ Unif(0.1, 0.3)
- 2,000 Examinees
  - Top and Bottom 10% of Norm(0, 1)
- for examinee in examinees :
- test.administer(examinee)

For the sake of time, won't show the code for creating the Items and Examinees. Once they're all initialized, to run the test, you simply call test.administer() on each examinee.

Results

	Max FI	KLI	a-Strat
First Two Correct			
Top 10%	-0.129	-0.144	-0.215
Bottom 10%	0.361	0.326	0.468
First Two Incorrect			
Top 10%	-0.858	-0.782	-0.611
Bottom 10%	0.105	0.102	0.204

Future of OSCATS

- More Item Models, Algorithms
- Closer integration with MATLAB
- [Bindings for R]
- Test Description XML Format
- GUI for Creating/Running CAT Simulations
- Item Parameter Estimation

- http://oscats.googlecode.com

Note: Some of the features demonstrated today are in the development version of OSCATS (code repository), so if you try to download the released version, it won't quite work like shown. But, a new release is planned to happen in the next few weeks. The complete code for this example will be included in that release.